1.3 Arithmetic Expressions - A First Look

*Standard operators (+, -, *, /)*

One of the major operations all computer programs have built in are arithmetic computations. These are used as parts of full statements, but it's important to understand the rules of arithmetic expressions in general, so that we can determine how the python interpreter calculates each expression. We can easily see the value of any arithmetic expression by typing it into the interpreter:

```
>>> 3+4
7
>>> 17-6
11
>>> 2 + 3*4
14
>>> (2 + 3)*4
20
>>> 3 + 11/4
5.75
```

Note that we'd never use any of these expressions as a whole line in a python program. The examples above are simply for learning purposes. We'll soon learn how to incorporate arithmetic expressions in python programs.

The four operators listed, addition (+), subtraction (-) and multiplication (*), work exactly as they were taught in grade school. As the examples above illustrate, multiplication and division have higher precedence than addition or subtraction and parentheses can be used to dictate which order to do operations.

1.4 Variables in Python

Idea of a Variable

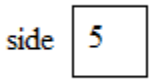
Part of the reason computer programs are powerful is that they can make calculations with different numbers, using the same set of instructions. The way in which this is done is through the use of variables. Rather than calculating 5*5, if we could calculate side*side, for any value of side, then we have the ability to calculate the area of any square instead of the area of a square of side 5.

Python makes variables very easy to use. Any time you want to use a variable, you can put the name of the variable in your code. The only caveat is that when you first create a variable, it does not have a well-defined, so you can't use that variable in a context where it needs a value.

The most simple way in which a variable can be introduced is through an assignment statement as follows:

```
>>> side = 5
```

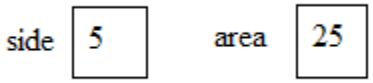
The name of the variable created is side, and the statement above sets side to the value 5. A picture of what memory looks like at this point in time is as follows:



If we follow this statement with

```
>>> area = side*side
```

Then our picture in memory is as follows:



Let's analyze what's happening here. Any statement with a variable on the left of a single equal sign and an expression on the right of that equal side is called an assignment statement. The goal of an assignment statement is to assign a variable to a value. It works in the following two step process:

1) Figure out the current value of the expression on the right, using the current values of the variables.

2) Change the value of the variable on the left to equal this value.

Thus, in the statement above, at the time it was executed, side was equal to 5. Thus, side*side evaluated to 25. Then, the box for area was replaced with the value 25.

Printing out the value of a variable

Of course, when we run these two statements in IDLE, we don't SEE any evidence that the variables are these two values. In order to do this, we need to learn how to print out the value of a variable in python. The most simple way to do so is as follows:

```
>>> print(area)
25
>>> print(side)
5
```

Notice that when we execute these prints, we DON'T include double quotes. Had we done the following:

```
>>> print("area")
area
>>> print("side")
side
```

the words in question would have printed instead of the values of those corresponding variables. What we see here is the previous rules we learned, that anything in between double quotes gets printed as is, except for escape sequences, does not change. Rather, a new construct (one without double quotes) must be used to print out the value of a variable.

Another natural question that arises is,"What if we want to print out both the value of a variable and some text in the same print?" Python allows us to do this by separating each item we would like to print with commas, as is shown below:

```
>>> print("The area of a square with side",side,"is",area)
The area of a square with side 5 is 25
```

If you carefully examine the text above, you'll see that between each item specified in the print (there are 4 items), python naturally inserted a space in the output, even though we didn't explicitly place one. This is python's default setting and in many cases is quite desirable. What if we wanted to place a period right after the 25 in the statement above? If we put a comma after area and place the string ".", we would find that a space would be inserted in between the 5 and the period:

```
>>> print("The area of a square with side",side,"is",area,".")
The area of a square with side 5 is 25 .
```

To avoid this, we can specify the separator between print items as follows:

```
>>> print("The area of a square with side ",side," is ",area,".",sep="")
The area of a square with side 5 is 25.
```

Notice that when we change the separator from a single space to no space, we then have to manually add spaces after the word "side" and before and after the word "is."

Not that the use of a different separator is typical, but any specifier can be used as the following example illustrates[1]:

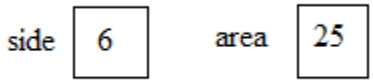
```
>>> print(side,side,area,sep=" }:-) ")
5 }:-) 5 }:-) 25
```

Increment statement

Consider following the original two statements in the previous section with the somewhat confusing statement:

```
>>> side = side + 1
```

In mathematics, it's impossible for a variable to equal itself plus one. In programming however, this statement isn't a paradox. By following the rules, we see that at the current time of evaluation, side is equal to 5. It follows that the right hand side of the assignment statement is equal to 5 + 1, or 6. The following step is to *change* the variable on the left (which happens to be side) to this value, 6. The corresponding picture is as follows:



To prove that this is in fact what has happened, execute the following statement:

```
>>> print("area =",area,"side =",side)
area = 25 side = 6
```

[1] Wikipedia claims that this emoticon means "devilish." I promise that I, in no way, promote "devilish" behavior however!

One key observation to make here is that area is STILL 25. It has not magically changed to 36 (the new value of side*side) after side was changed. Python only executes the commands it is given. Thus, if we wanted to reevaluate the area of a square with side 6, we'd have to recalculate the area as well.

After we change side, if we run the following lines of code again:

```
>>> area = side*side
>>> print(area)
36
```

We see that NOW, area has changed to 36, because we have specifically reevaluated side*side and stored this new value back into the variable area.

Rules for Naming a Variable

Clearly, a variable can't be named anything. Instead, python has rules for which names are valid names for variables and which aren't. In particular, the only characters allowed ina variable name are letters, digits, and the underscore('_') character. Furthermore, variables names can't start with a digit. (This rule is so they aren't confused with numbers.)

In general, while it's not required, it's considered good programming style to give variables names that are connected to the function the variable is serving. In the previous example, both side and area represent the type of information stored in those respective variables. If the variables were named a and b, respectively, someone else reading the code would have much more difficulty figuring out what the code was doing. Many beginning programmers, due to laziness or other factors, get into the habit of creating short variable names not connected to the function of the variable. For small programs these programmers don't run into much difficulty, but in larger programs, it's very difficult to track down mistakes if the function of a variable isn't immediately apparent.

Two Program Examples

Using the set of statements above, we can create a stand alone program by typing the following in a separate window and saving it as a python program:

```
# Arup Guha
# 6/1/2012
# Python Program to calculate the area of a square.

side = 5
area = side*side
print("The area of a square with side",side,"is",area)
```

Executing this program leads to the following output:

```
The area of a square with side 5 is 25
```

Comments

Large pieces of code are difficult for others to read. To aid others, programmers typically put some comments in their code. A comment is piece of a program that is ignored by the interpreter, but can been seen by anyone reading the code. It gives the reader some basic information. A header comment is included at the top of each different program. It identifies the author(s) of the file, the date the file was created/edited as well as the program's purpose. To denote a comment in python, just use the pound symbol (#). All text following the pound symbol on a line is treated as a comment by the interpreter. Although it's not shown above, a comment can start in the middle of a line:

```
area = side*side # Setting the area.
```

However, it's customary just to comment on a line by itself in most instances as follows:

```
# Setting the area.
area = side*side
```

Now let's look at a second program that calculates the total cost of an item with tax:

```
# Arup Guha
# 6/1/2012
# Python Program to cost of an item with tax.

item_price = 10.99
tax_rate = 6.5
total_price = item_price*(1+tax_rate/100)
print("Your total cost is $",total_price,".",sep="")
```

The output of running this statement is as follows:

```
Your total cost is $11.70435.
```

For now, let's not worry about outputting our result to two decimal places. We'll get to that later.